Documentation for RnSort.h and RnSort.c

Steven Andrews, © 2003
See the document "LibDoc" for general information about this and other libraries.

```
void sortV(float *a,float *b,int n);
void sortVdbl(double *a,double *b,int n);
void sortCV(float *a,float *bc,int n);
void sortVliv(long int *a,void **b,int n);
int locateV(float *a,float x,int n);
int locateVdbl(double *a,double x,int n);
int locateVli(long int *a,long int x,int n);
float interpolate1(float *ax,float *ay,int n,int *j,float x);
double interpolate1dbl(double *ax,double *ay,int n,int *j,double x);
float interpolate1Cr(float *ax,float *ayc,int n,int *j,float x);
float interpolate1Ci(float *ax,float *ayc,int n,int *j,float x);
void convertxV(float *ax,float *ay,float *cx,float *cy,int na,int nc);
void convertxCV(float *ax,float *ayc,float *cx,float *cyc,int na,int nc);

void setuphist(float *hist,float *scale,int n,float low,float high);
void data2hist(float *data,int dn,char op,float *hist,float *scale,int hn);
double maxeventrateVD(double *event,double *weight,int n,double sigma,double
     *tptr);
```

Requires: `<math.h>`,`"Rn.h"`
Example program: `SpectFit.c, LibTest.c`

History: Written 12/98. Works with Metrowerks C. Moderate testing. Added complex function 2/02. Added some double precision routines 10/02. Added `maxeventrateVD` 9/05. Added `sortVliv` and `locateVli` 11/29/06. Added `setuphist` and `data2hist` 12/06.

These routines work on sorted arrays of floats or doubles, and may be used in close conjuction with other matrix and vector routines. For virtually all routines, the sorted array is assumed to be in ascending order.

Some functions are for the use of histograms or other specialized lists of sorted numbers.

Functions

```
void sortV(float *a,float *b,int n);
```
`sortV` sorts vector `a` in order from smallest to largest value. The routine first checks for a forward or backward pre-sorted vector, and then, if neccessary, does a heap sort using a routine nearly identical to that given in *Numerical Recipes*. `b` is rearranged in the same manner as `a`, but does not influence the sorting in any way. `b` may be either `NULL` or the same vector as `a`, if a dual vector is not required.

```
void sortVdbl(double *a,double *b,int n);
```
`sortVdbl` is identical to `sortV` except that all numbers are in double precision.

```
void sortCV(float *a,float *bc,int n);
```

sortCV is identical to sortV except that the required vector bc is a complex vector, with 2n elements that alternate real and imaginary components. a is still real.

```
void sortVliv(long int *a,void **b,int n);
```
This is identical to sortV except that a is an array of long integers and b is an array of void*s. Also, b is required.

```
int locateV(float *a,float x,int n);
```
locateV locates the largest element of a that is smaller than or equal to x, where a is a sorted array. It returns the index of that element. If x is smaller than any value in a, then –1 is returned; if x is larger than any value in a, then n-1 is returned. If several elements of a are equal to each other and x is either equal to them or is between their value and the next higher value, then the element of the collection with the highest index is returned. This routine uses a bisection type routine, which is almost exactly copied from *Numerical Recipes*.

```
int locateVdbl(double *a,double x,int n);
```
locateVdbl is identical to locateV except that all numbers are in double precision.

```
int locateVli(long int *a,long int x,int n);
```
This is similar to locateV. Differences are that all numbers are long integers and the function only returns the index if an exact match is found. Otherwise, it returns -1.

```
float interpolate1(float *ax,float *ay,int n,int *j,float x);
```
interpolate1 does linear interpolation at position x. ax and ay are input *x* and *y* vectors, where the *x* values are sorted in ascending order, and n is the number of elements in each vector (minimum of 1). The routine needs the closest smaller *x* value. If its index or an index close by, but below it, is known, then send that value in as *j. If the index is not known, then send in *j as -2 and the routine will locate it with locateV. Either way, the correct index is returned in *j (identical to locateV). If multiple elements of ax have the same value: if x equals that value, the corresponding ay with the highest index is returned, if x is less than them, the one with the lowest index is used for interpolation, and if x is greater than them, the one with the highest index is used for interpolation. Here is a typical code fragment using interpolate1, which takes a sorted data set and rewrites it using a different set of *x* values.

```
for(j=-2,i=0;i<n2;i++) y2[i]=interpolate1(x1,y1,n1,&j,x2[i]);
```

```
double interpolate1dbl(double *ax,double *ay,int n,int *j,double x);
```
interpolate1dbl is identical to interpolate1 except that all numbers are in double precision.

```
float interpolate1Cr(float *ax,float *ayc,int n,int *j,float x);
```
interpolate1Cr is identical to interpolate1 except that the vector ayc is a complex vector, with 2n elements that alternate real and imaginary components. The function returns the interpolated value of the real components.

```
float interpolate1Ci(float *ax,float *ayc,int n,int *j,float x);
```
interpolate1Ci is identical to interpolate1Cr except that the function returns the interpolated value of the imaginary components.

```
void convertxV(float *ax,float *ay,float *cx,float *cy,int na,int nc);
```

convertxV takes a sorted *x*,*y* data set in ax and ay and interpolates the data to a different set of *x* values, from cx, outputing the result to cy. ax and cx should be sorted beforehand. This just does what the code fragment above shows, except that this routine is a little faster and easier to use. Also, it checks first to see if all terms in cx are equal to those in ax, in which case the data are copied directly. n needs to be at least 2.

```
void convertxCV(float *ax,float *ayc,float *cx,float *cyc,int na,int nc);
```
convertxCV is identical to convertxV except that ayc and cyc are complex vectors with 2na and 2nc elements respectively.

```
void setuphist(float *hist,float *scale,int n,float low,float high);
```
Sets up arrays for a histogram. hist will contain the histogram data, and has size n. scale is the histogram scale, which also has size n. The histogram will be set up for the range from low to high, plus a bin at each end for values that are below low and that are above high. For example, consider 5 bins from 0 to 10 in steps of 2: bin 0 is for $(-\infty,0)$, bin 1 is for [0,2), bin 2 is for [2,4), …, bin 5 is for [8,10), and bin 6 is for $[10,\infty)$. This would be setup with low as 0, high as 10, and n as 7. Both hist and scale would need to be pre-allocated to size 7. hist would be returned with all 0s and scale would be returned with the maximum value for each bin: 0, 2, 4, 6, 8, 10, FLT_MAX.

```
void data2hist(float *data,int dn,char op,float *hist,float *scale,int hn);
```
Takes an unsorted list of dn data values in data and sorts them into histogram bins in hist. If op is '=', any prior histogram counts are cleared from hist; if op is '+', these data are added to any prior counts; and if op is '-', these data are subtracted from any prior counts. There are hn total histogram bins with upper bounds given with scale, exactly as defined as in setuphist.

```
double maxeventrateVD(double *event,double *weight,int n,double sigma,double
*tptr);
```
maxeventrateVD inputs an unsorted vector of event times in event, which has n elements and has respective weights in weight. It finds the time at which the rate of weighted events was highest, where this is the time that maximizes $\sum_i w_i/[\sigma\sqrt{(2\pi)}]$ $\exp[-(t-a_i)^2/(2\sigma^2)]$. Send in weight equal to NULL if all events are to be weighted equally. This is a Gaussian smoothing of the delta functions that represent the events with standard deviation σ = sigma. The rate is returned. If tptr is not input as NULL, the time of the maximum rate is returned in tptr. Note that the maximum event rate depends on σ. This scans the whole event list and then does two more scans over progressively narrower regions. A faster but less reliable algorithm would use a binary search for the zero of the derivative.